POOSC '99

Object Oriented Concepts for Parallel Smoothed Particle Hydrodynamics Simulations

Stefan Hüttemann[1] Michael Hipp[1] Marcus Ritt[1] Wolfgang Rosenstiel[1]

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Arbeitsbereich für Technische Informatik
Sand 13, 72076 Tübingen
e-mail: {hippm,hutteman,ritt,rosen}@informatik.uni-tuebingen.de

Abstract. In this paper we present our object oriented concepts for parallel smoothed particle hydrodynamics simulations based on a 3 year work experience in a government funded project with computer scientists, physicists and mathematicians.[1]

In this project we support physicists to parallelize their simulation methods and to run these programs on supercomputers like the NEC SX-4 and Cray T3E installations at HLRS Stuttgart (www.hlrs.de).

First we introduce our portable parallel (non object oriented) environment DTS. Benchmarks of simulations we parallelized are shown, to demonstrate the efficiency of our environment.

Based on these experiences we discuss our concepts developed so far, and future ideas for object oriented parallel SPH simulations at two different layers. An object oriented message passing library with load-balancing mechanisms for our simulations at the lower level, and an object oriented parallel application library for our physical simulations on the upper level.

1 Motivation

In a collaborate work of physicists, mathematicians and computer scientists, we simulate astrophysical systems. In this paper we present our object oriented concepts based on our experiences made in a government-funded project[1] in the last three years. Smoothed Particle Hydrodynamics (SPH) is the method used by the astrophysicists to solve a Navier-Stokes equation (see [5], [8]). SPH became widely popular in the last years. SPH is now also used as an alternative for grid based CFD simulations (e.g. in automobile industry).

The astrophysical problems are open boundary problems of viscous compressible fluids. SPH uses particles that move with the fluid instead of grid

[1] SFB 382: "Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern" (Methods and algorithms to simulate physical processes on supercomputers)

points as in other CFD simulation methods. This makes SPH much more difficult to parallelize than grid based CFD methods.

1.1 Portable Environment

Programming with threads showed to be well-known and simple enough to serve as a basis for a portable parallel programming environment. The first approach, named "Distributed threads system" (DTS) [2] generalized the notion of threads for distributed memory machines. A compiler was written to simplify the task of identifying and creating parallel threads.

A major drawback of this system was its pure functional programming model: the communication between threads running on different nodes was not well supported. We are investigating, whether a combination of (global) threads and distributed shared memory, i.e. a logical consistent memory for machines with physically distributed memory, is suitable for our applications.

Currently we are working at the task of combining the basic ideas of this system, namely distributed threads and shared memory, with object-oriented concepts. This system, written in C++, is described in section 3.

For Cray T3E we used another approach. We provide a high level application interface optimized for SPH-like programs. The library has a simple to use interface and hides near all of the parallelization and explicit communication. A programmer only has to give some hints to optimize the communication and load balancing. The library itself is based on the native SHMEM message passing for Cray T3E or alternatively on MPI.

1.2 The need for Object-Oriented techniques

There were reasons to redesign our simulation environment using object-oriented techniques:

1. using message objects on the lowest level to communicate between concurrent program units on distributed memory computers seems to be most natural and easy to use for our simulation methods.
2. the growing complexity of our simulation programs requires structural elements in the programming paradigm not offered by e.g. FORTRAN or C. Also using an object-oriented approach to describe the problem is closer to the physical model used.
3. to exploit the usual features promised by object-oriented programming (reusability etc.) our project partners tried programming in C++; which resulted in anything but reusable, modular software. It showed, that just by switching to C++ physicists do not gain much, and fall back to FORTRAN-style programming.

   Our goal is to provide a well documented library of reusable and extensible solutions for astrophysical simulation methods. This also should give a guideline on how to use object-oriented techniques for our simulation methods (e.g. SPH).

2 Parallelizing SPH

We could gain a lot of experience with two different types of parallelization of the SPH code for shared memory machines based on DTS and for machines with distributed memory for which we developed a portable procedural library.

Both implementations can compete with other SPH codes for parallel machines such as the codes for Cray T3D, CM-5 or Intel Paragon [6,4].

2.1 Shared Memory Machines

The SPH code was implemented on the NEC SX-4 using DTS. The usage of DTS allows to run the same SPH code both on the NEC SX-4 and on other machines[1] without modifications. The parallel SPH code was used for benchmarking, using different numbers of CPUs. The results prove the high quality of the parallelization features of the NEC SX-4.

The right side of Figure 1 shows the parallel efficiency of the parallel SPH code on the NEC SX-4. For 10 000 particles the parallel efficiency decreases from 90% on two CPUs to 60% on 20 CPUs. For 100 000 particles the parallel efficency is more than 90% for all 20 CPUs.

Further runtime improvement can be reached by vectorization of the code. In the left side of figure 1 we present a comparison of the runtimes of SPH simulations of the same test problem with different codes. Together, the measurements in figure 1 show, that already using as less as 2 CPUs on the NEC SX-4 a runtime improvement compared to the optimal sequential SPH code can be achieved. As the efficency is excellent for 20 processors, very good runtime improvements can be expected using more CPUs.

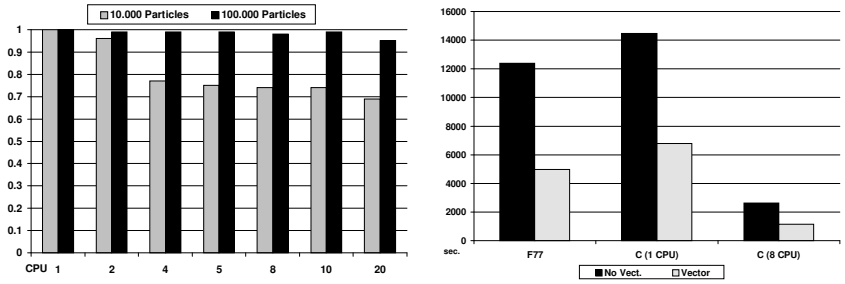

Fig. 1. *Left:* parallel efficiency on NEC SX-4. *Right:* Vectorization and F77 solution of a SPH simulation with 10 000 particles for optimized F77 sequential code, parallel SPH on 1 CPU and parallel SPH on 8 CPUs.

[1] e.g. SGI Onyx2, HP V-Class

2.2 Distributed Memory Architectures

Besides the DTS SPH code for shared memory machines, there are a few implementations for SPH on parallel machines such as the PTreeSPH [3]. The PTreeSPH code is based on MPI and, therefore, is portable to nearly every platform. We decided to go another way, because we see the need for a more efficent implementation on some architectures. We developed an abstraction layer optimized for SPH like problems with two different low level implementations for the communication.

The slower portable implementation is based on MPI. The other implementation is based on the Cray SHMEM message passing library, which provides functions oriented at the Cray T3E hardware capabilities and therefor gains a better performance.

Having two different implementations allowed us to test the flexibilty of our SPH abstraction layer. We first wrote the SHMEM implementation and specified the interface on which we put our physical code. It showed that the layer was flexible enough to later add a MPI implementation without changing the interface or the physical application.

Parallelization An essential idea for parallelization was to use two different domain decompositions depending on the type of computation:

1. All computations without neighbor interaction are done on an equally sized subset on every node. The subset is selected by splitting the particle field into n parts for n nodes. A node also operates as a *relay* node for its subset. Information about a specific particle can always be found on its *relay* node.
2. For computations with neighbor interaction all particles are sorted according to their positions into a grid with equally sized cells. These cells are assigned to nodes in a way that every node holds the same number of particles.

The load balancing is good in both cases, because the computation takes about the same time for every particle. For the case of very unbalanced computations we have the option to do *load stealing* between nodes to optimize the load balancing.

This approach reduces communication overhead and memory comsumption because the particle information resides on one node as long as possible. Also, the computation of this domain decomposition is fast enough to be done *on the fly*. This is important, because the particle positions change after every integration step.

Native SHMEM communication vs. MPI We measured the performance of the MPI code on the Cray T3E in Stuttgart and on the IBM SP system in Karlsruhe. On the SP we used 128 P2SC thin nodes with 120MHz. The tests showed, that the MPI implementation of the Cray T3E is worse

compared to the native SHMEM library (see figure 2). Our tests show that Cray could easily improve the MPI performance by making better wrappers around existing SHMEM calls. For some communication parts, such as gather operations of large arrays, the throughput decreased from about 300MB/s using SHMEM to 120 MB/s using MPI. On the SP we achieved the expected performance of around 50MB/s. The whole code didn't perform this good on the SP system, because the implementation is optimized for Cray T3E and depends heavily on good communication bandwith and latency in order to scale beyond 64 Processors.

Results It proved to be beneficial to use an abstraction layer, which allows the exchange of low level parts without changing the application to obtain the best performance on a given hardware platform. In the previous projects this was a procedural abstraction layer. For our current developments we choose an object oriented programming model for this abstraction. The crucial point for the parallelization is a smart domain decompostion optimized for both, the machine and the problem. An object oriented modelling of domain decompositions, which we want to describe in a later chapter is therfor one of the requirements for our parallel environment.

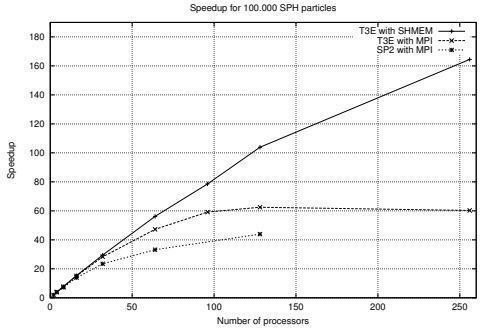

Fig. 2. Speedup of the SPH Simulation on Cray T3E with SHMEM and MPI and on IBM SP with MPI for a mid-size problem with 100 000 SPH particles. For larger node numbers the curves are dominated by the non parallelized communication parts such as gather/broadcast operations between all nodes. Please note the effect of the limitations of the MPI implementation on the Cray T3E beyond 128 nodes.

3 An object-oriented parallel runtime system

Based on the experiences with existing object-oriented parallel systems and our own object-oriented codes, we now describe our concepts for a object-oriented parallel runtime system.

There were numerous reasons, which motivated the redesign of these layers, despite of the existence of object-oriented message passing libraries like MPI++ or MPC++ [9]. The most important were the lack of thread-safe

implementations and the missing integration of modern C++ concepts like templates and the support for the standard template library.

To support object-orientation for parallel programming, we extended our model of parallel computing with threads on machines with distributed memory to C++ objects. In this model, an object – extended for architectures with distributed memory – is the basic entity of data communication. Objects can be migrated between different address spaces and replicated to improve performance. Migration and replication can be done explicitly by the user. For specific applications-domains, for example particle codes, we intend to provide tailored load-balancing components which free the user from explicitly specifying the data distribution. The methods for guaranteeing consistency are based on the well-known consistency protocols from distributed shared memory. In addition to this, objects support asynchronous remote method invocations. This corresponds to the asynchronous remote procedure call in our former approach, that is, a thread fork extended for machines with distributed memory, . Based on these facilities we plan to integrate some library solutions for a couple of common problems, for example automatic parallelization and load-balancing for independent data-parallel problems. These libraries will be application-independent (in difference to the higher-level libraries described later, which support specific physical problem domains like particle simulations).

To realize this model, we started implementing a basic layer for object-oriented message-passing. This layer can be used independently from the higher-level layers. To keep it portable, it is designed to be easily implemented on different low-level communication primitives. One implementation is on top of MPI to support a broad range of parallel architectures. There also exists an UDP-based version for test runs in a local environments without MPI support. Currently we are porting the library to the Cray T3E to run performance tests. Due to the bad MPI performance on the Cray T3E (see 2, we will also implement a native Cray SHMEM based version for production runs on this platform.

To simplify the migration from procedural codes written in MPI, the functions and methods are very similar to the MPI calls as far as suitable for the object-oriented interface. The main focus lied on extending the MPI functionality to objects without losing type-safety and the full integrations of the STL, i.e. transferring STL containers as well as using iterators for send and receive calls. To support the higher-level layers the library had to be programmed thread-safe.

The user interface for the object-oriented message-passing is straightforward with Communicator objects, send- and receive-methods. Composite objects like STL containers, arrays or user objects are decomposed into basic types by an overload resolution/traits technique [11]. Therefore, the user has not to deal with the unattractive concept of MPI datatypes, without loosing type-safety. To send and receive objects, the user has to provide serialize and deserialize methods, specifying how an object can be broken in components.

To minimize the communication overhead and prevent writing unnecessary serializer methods, objects with a trivial copy constructor can be handled directly by the library. We are also working an a code preprocessor which will generate the serializer methods for most objects automatically. Furthermore, note that the techniques used for sending objects and other C++ data types over the net, can be used without modification to implement persistent objects and application-level checkpointing.

A message-passing based version of an object-oriented SPH code will be our first test application. Based on these experiments, we plan to implement the higher-level layers by the end of this year. A portable object-oriented thread library will be integrated in the near future.

4 Towards object-oriented parallel SPH

4.1 Design Patterns

We cannot ignore the demand for programming in C or FORTRAN. To provide simply an implementation in C++ will not be accepted by our project partners. We had to find a way to write down our solutions in a "Meta-Language". Using Design Patterns serves this purpose best. We have an easy to understand way to document our solutions that is not bound to any programming language. Writing the design patterns in UML, we can use tools to implement the documented Design Patterns in e.g. C++ (almost) automatically.

As a first step towards an object-oriented SPH program, we used an easy to parallelize Monte Carlo simulation of the pulsar HER-X1. Looking at the problem as a programmer the Monte Carlo simulation and the SPH simulation are similar, because they are both particle simulation methods (in the case of the Monte Carlo simulation the particles are photons).

In the following we want to give an overview over the design patterns we used. The names for the patterns are taken from the Design Pattern book by E. Gamma (see [10]), but our patterns might differ from those in the book (we still need to give names to our patterns).

Composite Pattern for Domain-Decomposition Domain decomposition is the method used to parallelize particle simulations like SPH. The simulation area of all particles is decomposed into separate simulation-domains. The domain consists of particle lists that are being used to evaluate the equations. Since the particles interact, information must flow between the domains.

The main problem with the domains is the communication between the domains, and how to update the particle lists in each domain during the simulation.

To describe the solution for this problem, we used a pattern similar to the composite design pattern. The general behaviour common to all simulation-domains, the ability to communicate with other domains, is defined in the

abstract root-class *SimulationArea* (see fig. 3). A concrete simulation area will inherit the basic communication methods from this parent class, and change the implementation to its own needs.

Using this pattern, you can write simulation programs with compatible simulation subdomains without the need to rewrite the communcication between the different simulation-domains. To parallelize a simulation build us-

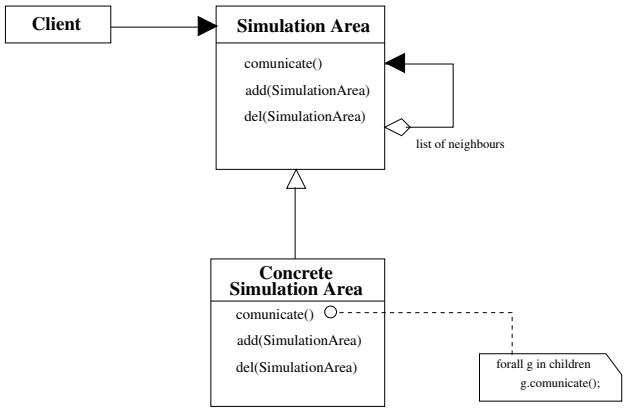

Fig. 3. Design Pattern used for Domain-Decomposition

ing the Domain-Decomposition, we make the *SimulationArea* root-class inherit from a class similar to a *Thread*-class, so that all simulation-area objects become *active* objects. (Active in the sense, that these objects run concurrently).

Iterator Pattern to step through neigbour-lists The iterator pattern is used to step through dynamic lists of neigbour simulation areas. This pattern is also available in C++ STL.

Strategy Pattern to select a numerical algorithm The core functionality of a simulation of physical processes always are some numerical algorithms. The selection of algorithms is stored in libraries for procedural languages like C or FORTRAN. To keep the advantage of a fine-grained selection of different algorithms we choose a strategy pattern for our numerical algorithms.

A strategy always works on a specific context. As an example we use the coordinate transformation of a vector (see fig. 4). Here the vector is the context of the strategy, and the different transformations are the concrete strategies, that have to be implemented for a simulation. The basic functionality is defined in the root-class for the coordinate transformation strategy. In this example the root-class contains the methods `toGeneric` and `toSpecial`. These methods transform the coordinates of a vector from and to a special coordinate system to a (pre-defined) generic coordinte system.

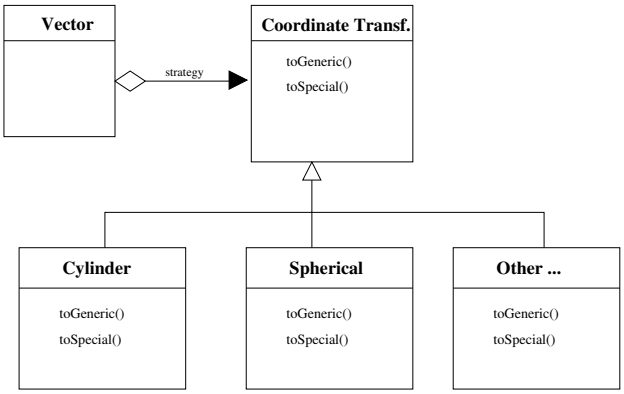

Fig. 4. Strategy Desing Pattern for Coordinate Transformations

Facade Pattern to handle input parameters To handle user input parameters independent of the objects used in the simulation, we use the facade pattern. A facade object collects all input, and marshalls the parameters to the correct object for this simulation.

Factory Pattern to create the particles A *Factory* pattern can be used to create the particles for the simulation. Communicating a single particle will not be efficient, therefore container classes for particles are necessary. Particles created using the *ParticleFactory* can now be collected, and put into container classes (see fig. 5).

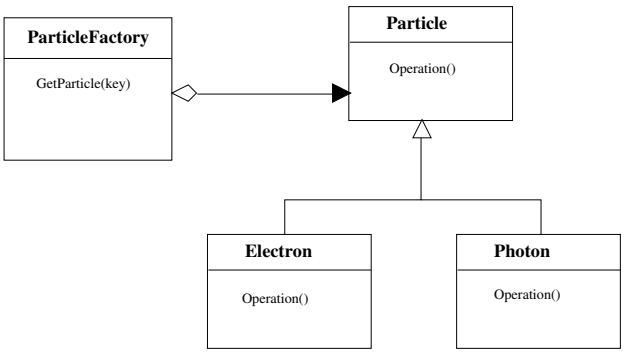

Fig. 5. Factory pattern for particles

Documentation Documenting the Design Patterns in a modern, easy to read way was achieved by using multi-frame HTML documents. Solutions that

are fun to read find generally better acceptance, even if there is no direct implementation in the favorite language of the programmer, e.g. FORTRAN. Also documenting the simplicity of our ready to use solutions motivates more physicists to take a look at a new programming paradigm (some even take a second look).

Prototyping in JAVA We also tried using JAVA for prototyping. Implementing our design patterns written in UML is fast to do in JAVA. The JAVA protoypes cannot be used for the real problem (in our case because there are no JAVA environments on Cray and NEC computers). The prototype is really just a prototype.

References

1. Allen, M. P., Tildesdley, D. J.: *Computer Simulation of Liquids.* Oxford University Press (1992)
2. Bubeck T.: *Eine Systemumgebung zum verteilten funktionalen Rechnen.* Eberhard-Karls-Universität Tübingen, Technical Report WSI-93-8 (1993)
3. Davé, R., Dubinski, J., Hernquist, L.: *Parallel TreeSPH.* New Astronomy volume **2** number **3** (1997) 277–297
4. Dubinski, J.: *A Parallel Tree Code.* Board of Studies in Astronomy and Astrophysics, University of California, Santa Cruz (1994)
5. Lucy, Leon B.: *A Numerical Approach to Testing the Fission Hypothesis.* Astron. J volume **82** (1977) 1013–1924
6. Warren, S. Micheal, Salmon, K. John: *A portable parallel particle program.* Comp. Phys. Comm. volume **87** (1995) 266–290
7. Bubeck, T., Hipp, M., Hüttemann, S., Kunze, S., Ritt, M., Rosenstiel, W., Ruder, H., Speith, R.: *Parallel SPH on Cray T3E and NEC SX-4 using DTS* High Performance Computing in Science and Engineering '98, Springer (1999) 396–410
8. Gingold, R. A., Monaghan, J. J.: *Smoothed particle hydrodynamics: theory and application to non-spherical stars* Mon. Not. R. astr. Soc. volume **181** (1977) 375–389
9. Wilson, Gregory V.,Lu, Paul: *Parallel Programming using C++* The MIT Press, Cambridge (1996)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: elements of reusable object-oriented software* Addison-Wesley (1995)
11. *Programming languages – C++. International Standard 14882.* ISO/IEC (1998)